The Report Committee for Ayberk Kanberoglu
Certifies that this is the approved version of the following Report

Survey on Graph Databases and Their Applications

APPROVED BY
SUPERVISING COMMITTEE:

Ying Ding, Supervisor

Amelia Acker, Co-Supervisor

Survey on Graph Databases and Their Applications

by

Ayberk Kanberoglu

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Information Studies

The University of Texas at Austin

August 2020

Abstract

Survey on Graph Databases and Their Applications

Ayberk Kanberoglu, MS Info. St.

The University of Texas at Austin, 2020

Supervisor: Ying Ding, Amelia Acker

The idea of representing information or logic through graphs extends back to 18th century by Leonard Euler. The graph theory is one of the fundamentals of mathematics but it hasn't been until the recent years where the graph-based data models started to get applied to various aspects of the information world. The relational database models have dominated the database industry for the last fifty years. The precedence of relational models can be attributed to their storage space efficiency, reliability and the created abstraction between the database and the user. Even though relational databases have succeeded for a long time, with the rise of web, big data and unstructured data, the need for new data storage models became apparent. Graph based database models solve most of the shortcomings of relational models and they are quickly gaining popularity among various industries. In this report I will analyze the history of database models starting from the relational models and compare these models to the newer NoSQL storage models focusing on graph-based storage. I will give examples of successful implementation of such storage models and finally I will talk about unique applications of these graph databases.

Table of Contents

List of Figures

List of Tables

Introduction

History of Graph Theory

Even though graph theory was discovered independently multiple times in history, Leonard Euler is the first known person to use it. In 1736 Euler settled a back then unsolvable problem named Konigsberg bridge problem by representing the problem as a graph and proving that the graph was not traversable. In this problem, Euler represented each land area by a point and each bridge between land areas as an edge hence creating the first graph in history as we know it (Harary, 2001).

What is a Graph Database?

A graph is composed of two elements. Nodes and edges. Edges connect between the nodes. In the case of a graph database, graphs are expressed in node-arc-node (subject-predicate-object) triples (Hurlburt, 2017). In most graph database models nodes represent physical or conceptual objects, mainly our fundamental data. Edges represent the metaphysical constructs that create the relationships between the nodes (Hurlburt, 2017). So intrinsically a graph database treats the relationships between the data as important as the data itself. This system of designing a database allows the database to have no predefined model and grow more naturally (Sasaki, 2018).

Why Graph Database Models?

The internet is believed to be one of the main inventions that increased the popularity of graph models because of its networked structure. After the internet, suddenly many types of information models like ecosystems, supply chain models and transportation models started to get interpreted as networks (Hurlburt, 2017).

One of the main reason graph databases have gained popularity in the recent years is that they support convenient querying when searching for data. In traditional relational database models, complex SQL queries might be required to gather data but the explicit relationship nature of graph databases overcomes this problem and makes it more intuitive for users.

Graph database models are valuable when the information about the data interconnectivity is as important or even more important than the data itself. One advantage of graph models is that they are more natural to model the data since the nodes and edges can hold relevant information and connectivity. Second advantage of graph models is that it allows users to express queries in a high level of abstraction because the users don’t need to know the relevant structure of the existing data (Angles, 2008).

Relational Models

In 1970 Edgar F. Codd published the paper “A Relational Model of Data for Large Shared Data Banks” and since then it became the industry standard on modeling large scale databases. In this paper Edgar F. Codd (1970) talks about the need for a data bank system in which the users need to be protected from the organization of the data. This new relational model allowed users to access the data using a querying language while having no information about the internal structure of the data.

In relational models, the tables are modeled as relations that contain a number of tuples or records. Each tuple in a relation contains the same number of fields and each field has a predefined type such as an integer or a string. Querying done on these relations, outputs relations themselves, meaning the structure of the output is a set of tuples (Curé, 2015). Generally speaking, relations, attributes, and tuples can be mapped to tables, columns, and rows in that order.

This is where querying languages like SQL (*structured query language*) came into play and created the abstraction between database design and the user. There are a couple reasons why relational models succeeded in the last few decades. First reason is that SQL is a very intuitive language with a relatively easy learning curve (Curé, 2015). Statements made in SQL are closely related to their meanings in English making it so that even a person with no SQL experience can understand what the query is trying to retrieve. On top of that, the widespread adoption of Relational Database Management Systems (RDBMS) makes it so that almost all of the existing systems use a subset of the standardized SQL and enables users to switch between these systems with relative ease. For instance, MySQL, Oracle and PostgreSQL are all different SQL implementations but all of their syntaxes are very similar.

One of the main features of relational database models is that they use a transaction model that supports ACID properties. ACID is an acronym for atomicity (A), consistency (C), isolation (I) and durability (D).

— Atomic transaction means either the whole transaction goes through or it is rolled back.

— Consistency means the database will be left in a valid state after the transaction.
— Isolation is when transactions don't interfere with each other.
— Durability ensures that after transactions are executed, they are permanently stored even in system failures.

ACID has been the reliability benchmark database models strived to achieve for last few decades and it is one of the main reasons relational models are considered very reliable (Jatana, 2012).

Contrary to its name, relational databases actually don't store the name of the relationships within the database. Foreign keys only act as connections for the tables but not necessarily for the relationships. This way of designing a database works well when the database relationships are relatively linear, for instance if the database relationships are mostly one-to-one, one-to-many or many-to-one (Hurlburt, 2017). By eliminating the need to express every single relationship between records, the relational models can save a great deal of space. Considering the relational model was created in 1970's when the digital storage was much more limited compared to now, it makes sense that relational databases got traction. Another advantage of using a relational database is that they work much more efficiently when processing large number of records. The reason is because graph-based databases need to examine each record individually to determine its structure whereas relational databases know the structure of the data before the query even processes.

SHORTCOMINGS OF THE RELATIONAL MODELS

Rise of Semi Structured Data

In the early 2000's the advancements on the internet technology, social networks and internet of things created an influx of semi structured and unstructured data. When we talk about unstructured data, we usually mean raw data like images or sound. Semi structured data on the other hand is neither raw nor strictly typed (Abiteboul, 1970). A good example of semi structured data can be found on many HTML files where the data

consists of some level of tags and anchors. If we try to parse this type of data and load it into a database, we might encounter data type irregularities because much of the data is human typed and doesn't hold the data type in itself. Another irregularity with HTML files we usually encounter is that there are large data gaps between instances. For example, a data source about used cars near Austin, Texas doesn't treat all the cars in the same manner. While some cars have more amounts of data associated with them, like their interior color and maximum seating, some cars have very little amounts of data.

Semi structured data also arises when we combine multiple data sources. When we look at various websites that display used cars, we can see their data formatting is very different from each other. For instance, some webpages use prefix 'M' for miles whereas some pages use 'mil' and all of the websites use varying formats for dates. When we combine these multiple sources, we might end up with multiple attributes that represent the same thing and many more records with empty attributes.

Relational models have not succeeded at representing semi structured data because relational databases have schemas that require the naming and type of the records to follow certain standards and SQL was specifically built to query on structured data (Jatana, 2012). On the other hand semi structured data is rather incomplete and inconsistent. When we add semi structured data in to a relational model we either have to do significant amount of data cleaning or we have to deal with a database that is not efficiently storing the data it holds and requires very complex SQL queries to be executed.

Lack of Horizontal Scalability

Even though relational models allow for very efficient vertical scalability through addition of hardware components like RAM, SSD and CPU, they are very inefficient in growing horizontally when new servers need to be added to grow the database. The main reason for the inefficiency is because of the difficulty of joining tables in distributed servers. To join tables distributed among cluster of computers, more often than not the

data on one of the servers need to be copied into another server. The movement of data through the network is slow and creates a bottleneck. The second reason horizontal scalability is hard to achieve is because the data maintenance becomes significantly more difficult in distributed servers. When a single table ends up being stretched into multiple servers, the table schema modifications require every single table to be modified among all of the machines that use the table.

Highly Connected Data

As we've talked before relational databases save storage space by not explicitly defining each single relationship between records. Instead, the use of foreign keys allows for table to table connections. This way of designing a database works well when the database connections are mostly linear, for instance one to one or one to many. If each record in table1 connects to a record in table2 and the connections are same, then relational databases work very efficiently. When the data starts becoming highly connected and complex, relational model requires the creation of many to many relationship tables. Even then, many to many tables can only represent one type of association. If there are unique associations between records of each table then there needs to be even more tables. Writing queries for such databases requires unreasonable amounts of join operations. This design defeats the purpose of relational models because the database ends up being neither efficient in space and time nor intuitive to understand.

Post-relational Models

Object Oriented Database Models

In the early 1980's the shortcomings of relational models on data intensive domains started to become more apparent. A lot of the fundamental engineering and computer science applications started use object-oriented programming concepts where the relations between data objects were much more complex than what could be expressed in a relational model (Angles, 2008).

Object-oriented models describe entities as objects that have states and behaviors. State of an object is the set of values associated with an object whereas behaviors are set of methods that can be applied within an object (Kim, 1990). These behaviors can modify the state of the object. A class is a blueprint of an object where it defines the set of attributes and behaviors a certain type of object can have. Objects can communicate with each other through passing messages to each other and causing each other to execute methods (Silberschatz, 1996). Graph models and object-oriented models are similar to each other in the sense that they both make use of graph structures. Object-oriented models use graph structures to express inheritance hierarchies. On the other hand, there are fundamental differences between the two models. First of all, O-O models view the world as complex entities that can communicate through method passing, whereas graph models view the world as network of relations emphasizing the relationships between the data (Wylot, 2018). The fact that object-oriented models use the class structure to define objects also means that it has inherent schema in its design whereas graph models have no schema.

NoSQL

The term NO SQL was first introduced in the late 2000's when the people working in the software development sector started to complain about the relational database systems and SQL because of its complex and difficult to manage nature. As the cost of digital storage decreased, the database systems started optimize more towards software engineer

efficiency rather than storage and the NoSQL movement gained a lot of popularity (Schaefer, 2020).

NoSQL databases consist of various data storage models like key-value based, document and graph stores, but all of them have some level of common characteristics shared between them. The first characteristic of a NoSQL store is that the data is *persistent* meaning the data is stored in non-volatile memory like SSD or HDD. This characteristic is same in both relational models and NoSQL models. Second notion of NoSQL is that it adopts a more flexible data model where there is no scheme. This design decision is very critical because it allows the database users to add as many novel data into the database without wasting space. For instance, in a relational model, adding numerous attributes to a certain record would mean the schema of the relation needs to be modified and this results in many NULL values to be added to the records that doesn't define the newly added attributes. In NoSQL databases we don't need to assign values to attributes we don't know or have (Curé, 2015). NoSQL databases use a lot of data replication within their models and hence increase query performance by not requiring to move data between networks. On top of these traits NoSQL's weak consistency transaction model allows for high availability of the data and low latency on queries (Davoudian, 2018).

NoSQL Graph Store

Most of the NoSQL data stores like key-value based and document stores deal with entities as binary values, rows in tables or documents. Graph stores are different because the entities are stored in nodes and the relationships are stored on edges. Graph stores use a traversing query approach that queries data by following the edges. Because edges directly represent the relationships, no joins are needed unlike relational models. Even though not using joins makes traversing efficient, the objective of finding a node to start the traversal among millions of nodes is not trivial and requires unique solutions.

Graph models allow complex queries to be expressed easily. Query workloads can be divided into two categories*: online graph navigations* where a small part of the graph

is accessed and *offline analytical graph computations* where a significant fraction of vertices and edges are accessed (Davoudian, 2018). There are two types of online queries, these are path queries and pattern matching queries. Path queries look for paths that connect two nodes. For example, in a social network database we want to find the names of all of the people in Bob's friend network. This means that we want to find paths that start with a triple such that <Bob, isFriend, x> where x is the name of the friend in the network. Since we are looking for names in the whole friendship network, the traversal continues after finding a friend of Bob and tries to find friends of friends of Bob and so on. A pattern matching query on the other hand tries to find subgraphs that are isomorphic to a given pattern. Isomorphic graphs mean that two graphs are same in both structure and labels. An example could be finding all of Bob's friend who went to school in Texas. A pattern graph could be created for the query and the graph store can be traversed to find the exact such pattern. The three most popular graph query languages that have pattern matching capabilities are SPARQL, Neo4j Cypher and Gremlin (Davoudian, 2018). While SPARQL is designed to query RDF graphs, Cypher and Gremlin are designed to query property graphs.

Graph Database Models

OBJECT EXCHANGE MODEL

Object exchange model (OEM) is one of the earlier graph-like database models that was introduced in 1995. It is particularly good at representing semi-structured data and it gets its fundamentals from the object-oriented model. One of novel ideas of OEM was to not have an advanced structure that defined the values an object could have. Each object is essentially self-describing and consists of four elements which are *label*, *type*, *value*, and *object-id.* Label describes the object. Type defines what the type of the value is. Each object-id is a unique integer. There are two types of objects in this design. These are atomic objects and complex objects. Atomic objects consist of a value with a simple type like integers, strings or floats. Complex objects have a type of set where the value is a set of other object-ids. To give an example we might have an object labeled *color* with the value "red" and type string. Object-ID will usually be a unique integer identifier so let's say it is 24 for this case. This is an atomic object. Then we can create another atomic object labeled *max_speed (mph)* with value 185, type integer and Object-ID 32. Using these two atomic objects we can create a complex object labeled *car*. This object is going to have set as its type and the value for this set is going to be {24, 32} describing the two previously defined atomic objects. In this design if we think about each object as a node and each set of object-id's in a complex object as edges, we end up creating a simple graph as seen in Figure 1. OEM model creates a simple transition from object-oriented models to graph models where the design is much simpler where there are no classes, inheritance or methods (Papakonstantinou, 1995).

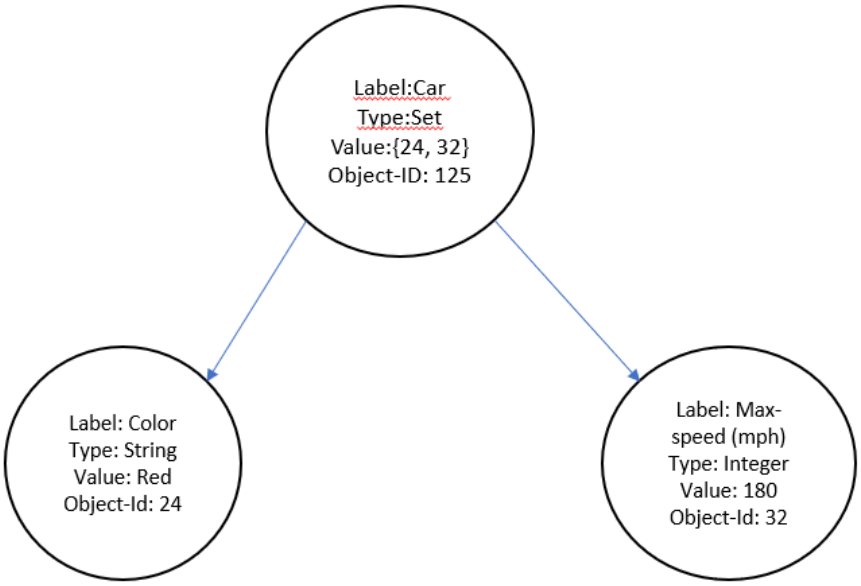

Figure 1 OEM Model as a graph

THE PROPERTY GRAPH DATABASE MODEL

The most common usage of graph database systems has been supporting property graphs. By definition property graphs are directed graphs (edges have direction from one node to another) where both the nodes and edges can have a set of properties (Angles, 2018). This set could be an empty set. Properties are usually represented as name:value pairs.

To give an example of a property graph design we can think of a triple such as "James knows Jill". One way to model this triple in a property graph is representing James and Jill as nodes and their relationship of knowing each other as the edge between the nodes. In this case James and Jill nodes can have properties such as their names, ages and ids. The "knows" edge can also have properties like "date_since" signifying the date in which the two people started knowing each other.

Property vs Relationship in Graph Database Design

One interesting design decision every property graph designer needs to make is whether to model something as a property or as a relationship to a separate node. This decision can have effects on the size of the database and efficiency of various query operations.

Consequently, the decision needs to be made depending on the expected types of query operations and their frequency.

To give an example let's think of a soundtrack database for a music streaming service. One way to design this database is creating separate nodes for each user and adding properties like name, email_address, age, and location to these nodes. Then we can create nodes for each soundtrack and add properties such as composer, genre, and length. Let's assume a soundtrack can have multiple genres hence the value in the genre property is a list of strings.

In this database design if a user decides to search for soundtracks that have two specific genres like “HipHop” and “R&B” then the query to find such soundtracks needs to traverse all of the soundtrack nodes and look for genre properties within these nodes that include both of the genre values. Such soundtrack database can be seen in Figure 2.

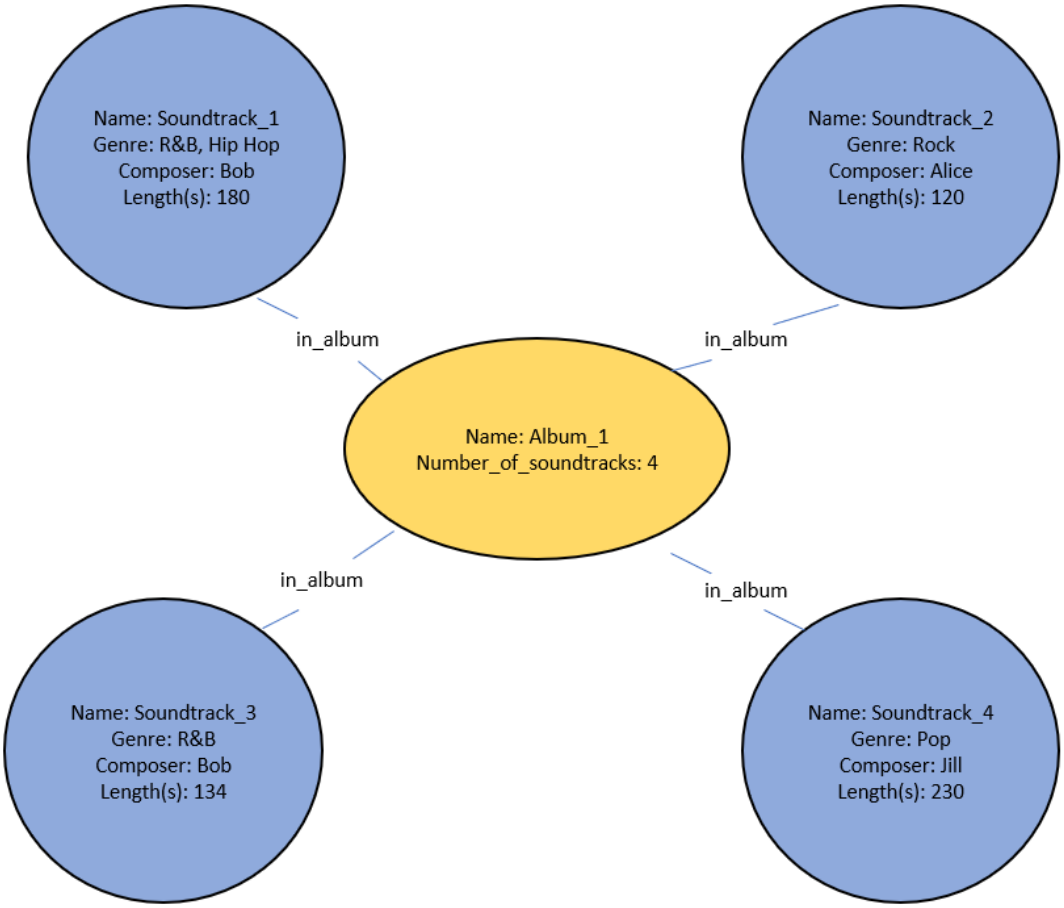

Figure 2 Soundtrack database with genre property

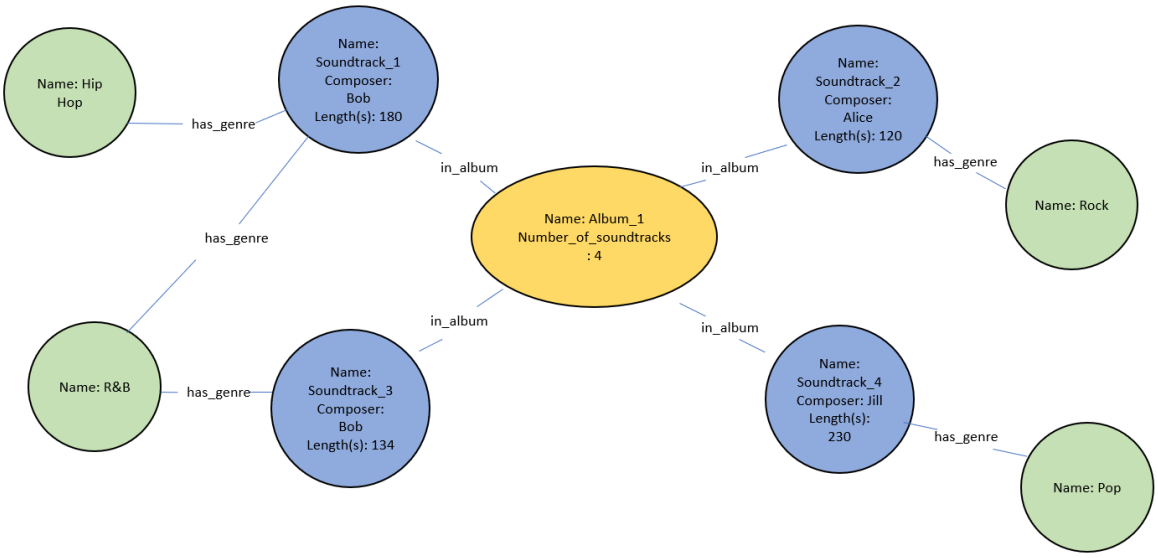

Figure 3 Soundtrack database with genre nodes

On the other hand, another way to design the database is having separate nodes for genres and make "has_genre" edge connection between the soundtrack nodes and genre nodes. In this scenario all of the soundtracks that have "R&B" as one of their genres will be connected to the same "R&B" genre node in the database. Now if we want query soundtracks that are both "HipHop" and "R&B" then we can simply look for one of the genre nodes (either one is fine) and traverse soundtracks that have connection to the other genre. This kind of database design will create new relationships between soundtracks and genres and make the database more complicated but on the other hand genre specific search functionalities will be significantly more efficient.

Similarly, properties such as composer and performer can also be refactored out as nodes in the database and this will allow for efficient composer level queries. At the end, the database design might end up with almost no properties within their nodes but have all connections between nodes that signify relationships.

RESOURCE DESCRIPTION FRAMEWORK (RDF)

As I've talked in one of the previous chapters, internet played a huge role in increasing graph-based databases popularity. When internet was first started in the 1990's, web was mostly consisted of interlinked documents. Even though we could trace the links between webpages, there was no relationship knowledge to be gained from the link connections. In 1999 World Wide Web Consortium (W3C, 2014) recommended a stack of technologies to support the growth semantic web and our understanding of the information on the web. The lowest layer of this technological stack was designed to create a global identification standard for the resources on the web. This global web resource standard was called *uniform/internationalized resource identifiers (URI/IRI).* IRI's could identify resources such as documents, images, objects, people or even concepts. Couple stacks above the URI, W3C defined the Resource Description Framework (RDF). RDF is a graph-based model where resources are linked to each other in the form of triples. These triples consist of subjects, predicates and objects. In RDF, triples express the directional relationship from subject to the object defined by the

predicate (Wylot, 2018). Each resource in the RDF model can be a part of numerous triples and take the role of subject, predicate or object depending on the triple. For instance, a subject in one triple can be a predicate on another triple. This behavior allows RDF to grow by linking various datasets on the web and create linked data.

To give an example the wikidata.org identifies Albert Einstein with the IRI: https://www.wikidata.org/wiki/Q937. Dbpedai.org identifies Nikola Tesla with the identifier http://dbpedia.org/page/Nikola_Tesla. Since IRI's are global identifiers they can be used in various different places. For example, www.schema.org presents various generic IRI's that could be used as predicates. One of these IRI's is http://schema.org/knows. This predicate expects both the object and subject to be of 'person' type and expresses that these two people know each other. Combining the three different IRI's gathered from three different resources we can create a triple such as:
<https://www.wikidata.org/wiki/Q937><http://schema.org/knows>
<http://dbpedia.org/page/Nikola_Tesla>.

One of the elements of RDF graphs are *literals*. Literals are basic values that are not IRI's (W3C, 2014). Literals could be strings, numbers, dates or some other data type. They are only allowed to be on the object part of a triple.

Blank nodes are type of nodes that don't contain a global identifier. They are local nodes that are not defined by IRI's. A good analogy would be thinking about blank nodes as simple variables from algebra such as x or y. We can define a blank node in an RDF graph and create triples with this blank node being either the subject or the object. Blank nodes are useful when the object or the subject we are trying to define doesn't have a global identifier. For instance, W3C RDF 1.1 primer gives the example that if we want to define the cypress tree that contains in the background of the painting Mona Lisa, we can use a blank node to identify it and then create triples to define that it is included in the painting and it is a cypress tree.

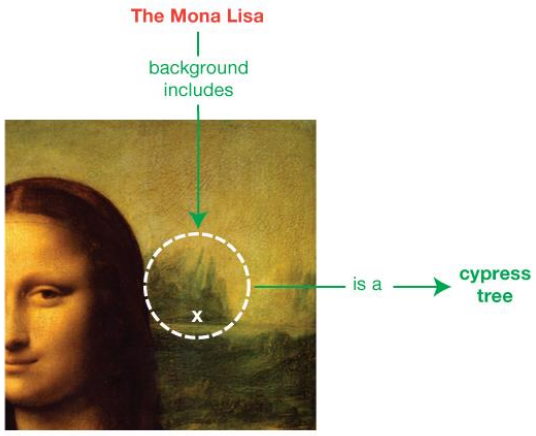

Figure 4 Sample Blank Node

SPARQL for RDF Querying

SPARQL has been recommended by W3C as the standard language for querying RDF data. SPARQL's main querying mechanism is graph pattern matching. SPARQL queries consist of three parts. These are pattern matching, solution modifying and output. Pattern matching is the process of searching the graph to match input pattern graphs. Some of the features included in this part are pattern unions, filtering possible values of matches and choosing the data source to find a possible match. Solution modifiers allow the queried values to be modified according to some criterial like limits, projections and distinct values. Lastly output can be shown in many forms like true/false values, new constructed RDF graph data or description of the resources (Perez, 2006).

Triple patterns in RDF are very similar to regular triples except one or more elements of triple patterns can be replaced by variables. A set of triple patterns is called a *basic graph pattern* (BGP) and a query consisting of only basic graph patterns is a BGP query (Wylot, 2018). BGP's can take various shapes. Figure 5 shows a few of these

shapes.

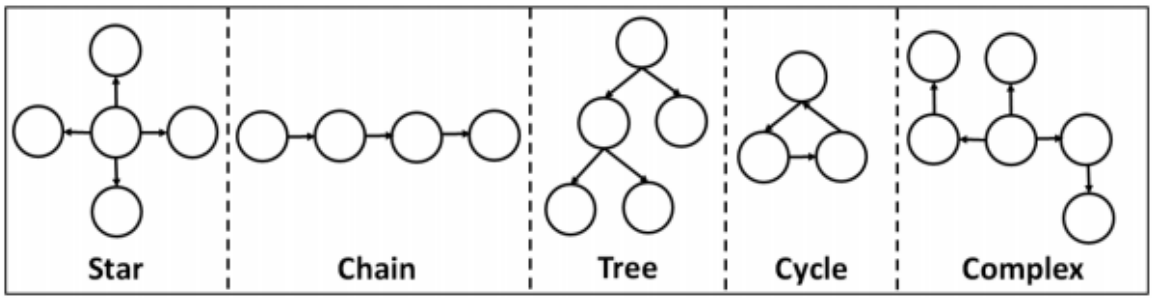

Figure 5 Sample Graph Patterns (Wylot, 2018)

SPARQL's query language syntax has adopted wide similarities to SQL to increase the languages adoption rate. The standard SELECT <template> FROM <source> WHERE <query pattern> query type that has been made widely popular by SQL is used by SPARQL as well. The variables in SPARQL are identified by question mark symbol (?) followed by variable names (Curé, 2015). The variable names store the retrieved data from the query.

```
PREFIX ex: <http://example.com/terms#>
SELECT ?name
WHERE
{
  ?user ex:lastname "Smith";
        ex:firstname ?name;
}
```

Figure 6 Example SPARQL QUERY

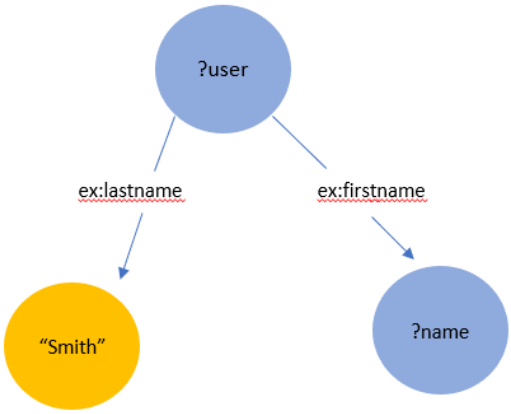

Figure 7 Sample Triple Pattern Graph

Let's assume we have a database of users. A very simple query we might want to execute could be finding all user's names whose last name is Smith. To achieve this

query first we need to define a variable to store the name and add it into the SELECT clause so the output exhibits this value. Then we need to write our RDF patterns in the WHERE clause. As we talked before RDF patterns are expressed as triples in the form of subject, predicate, object. First triple pattern we write creates a *user* variable and treats it as the subject of the triple. The predicate of the first pattern is ex:lastname. Ex in this scenario is a prefix for the predicate library stated at the top of the code. Prefixes allow for more concise code in SPARQL. Lastly the object of the triple is a literal specified as "Smith". This first pattern in SPARQL tells the query engine that find me a triple where the subject node connects to a string literal named "Smith" through the lastname predicate. The user variable will hold the label for the subject node of the triple which may or may not be the name of the person. The second triple is similar to the first one but main difference being that the object this time is also a variable named *name*. Combining these two triples we end up with a triple pattern as seen in Figure 7. Then the query engine tries to match this pattern in the graph and loads the data into variables but only displaying the variables in the SELECT statement.

Graph Database Stores

Graph database implementations use varying storage options for their backends. Some implementations use a *native* approach which means that they build their own storage backend and *non-native* means that the graph database system uses an existing storage for its backend like relational storage (SQL), document stores or object-oriented stores. Native storage allows for storage systems to be built specifically for graph structured data which usually ends up being more efficient but requires more resources on the development process. More specifically native storage options allow for constant time access to nodes and relationships.

NEO4J

Neo4j is the industry leader in the graph database stores. It is a native graph store with ACID based transactions which guarantees the reliability of the database. It uses a property graph storage model. Neo4j has created its own programming language called *Cypher* for managing the database store. Syntactically Cypher is similar to SQL. Its main goal is allowing users to create graph search patterns easily and intuitively. Cypher is open-source which makes it a suitable programming language to be adopted by larger audiences that want to implement their own graph database stores. Example Cypher queries that create and search nodes can be seen in Code Snippet 1.

```
// Create person node in the graph
CREATE (p:Person)-[:LIKES]->(t:Technology)
// Match without needing to specify direction
MATCH (p:Person)-[:LIKES]-(t:Technology)
```

Code Snippet 1 Cypher Code

The true strength of Cypher and Neo4j can be displayed best when we compare Cypher and SQL queries to each other. For instance, let's think about an academic institute database where there are tables for students, courses, and course registrations. Course registrations is a many to many relationship table where each record shows a student and a course pair in which the student takes the course. If we want to write a query that retrieves all of the course names a particular student has taken, it might look something like in Code Snippet 2. If we want to write the same query in Cypher it looks like in Code Snippet 3. The main difference between the queries is the succinctness of Cypher compared to SQL. In a graph database query system, there are no joins and the query itself explicitly tells what it is retrieving. On the other hand, SQL queries can be hard to understand and they are usually verbose.

```
SELECT Courses.name
FROM Courses
JOIN CourseRegistrations on Courses.id = CourseRegistrations.course_id
JOIN Students on Students.id = CourseRegistrations.student_id
WHERE Students.name = "John Smith"
```

Code Snippet 2 SQL query academic institute

```
MATCH (s:Student {studentName:"John Smith"})-[:TAKESCOURSE]->(c:Course)
RETURN distinct c.courseName;
```

Code Snippet 3 Cypher Query academic institute

One of the strong features of Neo4j is the ability to distribute data into distributed servers through Neo4j Fabric. Neo4j Fabric is a database sharding solution that allows larger graphs to be broken down into smaller graphs and store them in various databases ("What is a graph database", 2020). This systems design structure works by creating a coordinating database named *Fabric*. This database can be thought as the proxy database and it acts as the entry point for all of the queries made into the distributed servers. Fabric takes in queries from clients and routes the requests to the sharded databases which return the query results back to the Fabric. Then Fabric takes all of the results and applies the necessary filters or aggregations and finally returns the resulting data back to the client. Overall Neo4j Fabric shifts much of the responsibility of managing a distributed database server from software developer to database itself.

ARANGODB

ArangoDB is another native database system that supports graphs. Unlike Neo4j it is free and fully open source. The main feature of ArangoDB that separates it from other graph database systems is that ArangoDB is a native multi-model database. Native multi-model database system means that the database can support various data models like graphs, key-value stores, documents and SQL at the same time in a single system with only one query language ("What Is a Multi-Model Database", 2020). The main benefit of supporting various models at the same time is that software developers don't have to force fit new data into an existing data model or they don't have to create new databases

for the same software project because new data has different characteristics. By supporting various models in the same database, the overhead of managing multiple databases is cut down.

The fundamental storage system format in ArangoDP is JSON. The document modeled data is stored in JSON files where the document id is the key and the values are JSON documents. The graphs are modeled by creating a JSON document for each vertex and edge. The edges have special fields named "_to" and "_from" where they define the nodes it is connecting by their id's.

ArangoDB developed its own programming language named ArangoDB Query Language (AQL). AQL is a declarative programming language that is similar to SQL. It supports modifying and reading data but it does not support managing the database meaning it can't drop or create databases and collections etc. To illustrate AQL lets go back to our academic institute example. To retrieve all of the courses "John Smith" is taking, the query in AQL would look like in Code Snippet 4. First of all, in AQL the depth of the graph search needs to be specified. In this example it is 1 because we are looking for courses, students are directly taking. In the code snippet this part is specified as 1..1. If the graph search had a depth of 4 we would specify it by 1..4. Second part is we need to express whether the graph search is in the direction of the edges meaning "OUTBOUND" or in the opposite direction "INBOUND". Next the starting node is specified. In this example the starting node is labeled "JohnSmith" who is a student. Lastly the name of the graph is given which is "AcademicInstitute". After the graph search starts, we want to look for edges that indicate that the connecting node is a course taken by the student. In this graph these edges are labeled as "course_taken" and we only filter these edges. Since the query is written to get the names of the courses the student is taking, the return statement returns the "_key" of

the connecting vertex. Compared to SPARQL and Cypher, AQL is a much more procedural looking language. Instead of queries looking like graph patterns, they resemble more graph traversal algorithms which can be beneficial to some software developers.

```
FOR vertex IN 1..1 OUTBOUND "students/JohnSmith" GRAPH "AcademicInstitute"
        FILTER vertex.edges[0].label = “course_taken”
        RETURN vertex._key
```

Code Snippet 4 AQL code academic institute

ORIENTDB

OrientDB is similar to ArangoDB in the way it is also a multi model open source database that supports key-value, document, graph and object stores. It was written in 2010 by Luca Garulli in the programming language Java. The underlying data storage is JSON files. Unlike other graph database stores OrientDB uses SQL and adds some extensions on the language. The reason SQL was chosen is because it is the most widely recognized querying language in the world. The biggest difference between the regular SQL and the SQL used by OrientDB is that there are no joins in OrientDB and instead relationships are represented by links. Links are supported by the dot (.) notation. For instance, the academic institute example would be represented in OrientDB as in Code Snippet 5. Interesting thing about this query is that even though the retrieved data is from the course node, the FROM clause only specifies the student. This is because the SELECT statement matches the “course_taken” link to the students found and returns the names of the courses.

```
SELECT course_taken.name FROM Student WHERE name = “John Smith”
```

Code Snippet 5 OrientDB SQL academic institute

Graph Database Applications

Graph Database Applications in Transport Information Systems

There are various classical and complex applications of graph database models where we can see the key features of the model really show its benefits. In the realm of classical applications, transport information systems is one area where graph databases can have great benefits. Transport information systems gather data from various vehicles, timetables, and maps and analyze this data to discover bottlenecks and efficient routes to present to the end users (Czerepicki, 2016). As the number of users increases, the demand for the database to efficiently execute these queries and applications increases as well. Relational models can't naturally reflect the transport systems and since most of the queries are related finding optimal travel routes, algorithmic path algorithms can't be applied without doing transformation on the existing data. Graph databases present a natural form of abstraction to finding optimal paths in a transport database.

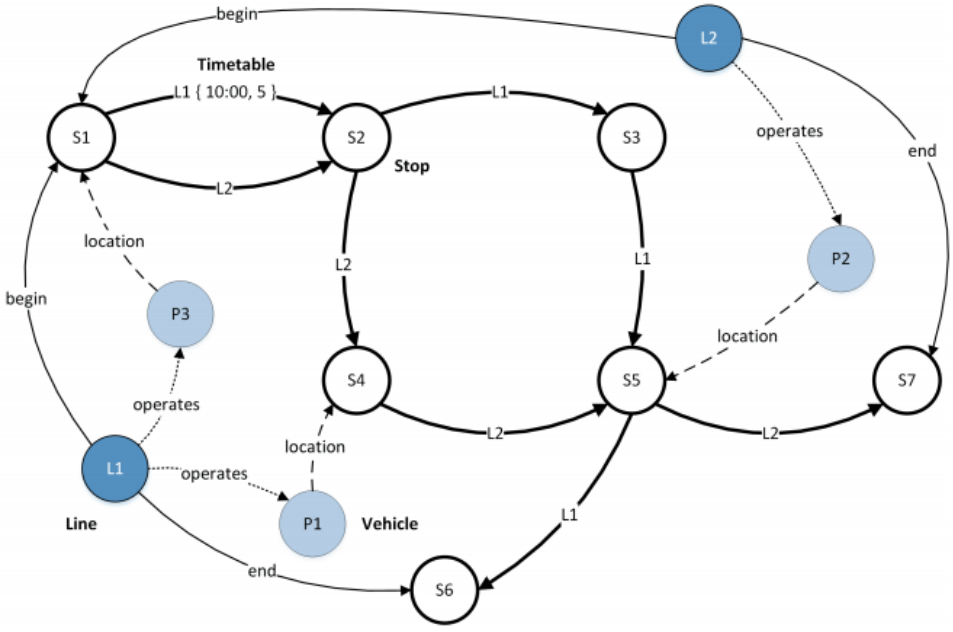

Figure 8 Sample Graph Database for Transportation System

Czerepicki (2016) explains a possible application model of a public transport system and what kind of algorithms can be applied on this graph database to efficiently query information. Initial creation of the model depends on transforming an existing entity-relationship model that represents public transport connections into a graph model. To achieve this transformation fundamental rules are that entities are interpreted as nodes, entity attributes are node attributes, and entity relationships are represented as edges.

Figure 8 demonstrates a sample graph database of a public transport system. Basic prefixes represented in this graph are stop (S), vehicle (P), and line (L), Creation of the graph starts by addition of these nodes with correct labels. Most of these nodes have specific attributes associated with them. For instance, stop nodes have GPS data as one of their property. Next step is creating connections between the nodes. In this specific example there are four different connections that can be created.

- Stop to stop connections are labeled by their *line* connection. A line connection has two attributes which are departure and time. Departure is the time of departure from the starting *stop* node S1 to ending *stop* node S2. Time attribute defines how long it takes for travel to complete.
- Line to stop connections define whether a certain stop is the start of a line or the end of it. This type of connection has no attributes other than its label.
- Line to vehicle connections define which vehicles operate on which lines.
- Vehicle to stop connections define the location of the vehicles.

Obviously since this is a graph database, given connections and nodes are not set and stone. In the future there might be more types of nodes, for instance nodes that define parking lots where cars might be residing when they are not at stops could be added.

One of the computations that is easy to execute in this model is finding all possible connections between two stops or finding the shortest amount of time required to go from one stop to another given the departure time. The main reason these computations are relatively easy in a graph database is that algorithms execute by

traversing the graph. If instead we used a relational database, we would need to create a data structure that represents a graph such as an adjacency matrix and execute the traversal on that structure. By using a graph database model, we skip the step of creating a graph structure and gain efficiency.

GRAPH DATABASES FOR KNOWLEDGE HANDLING

Knowledge handling is one of crucial pillars of artificial intelligence. When knowledge grows in quantity, it becomes very hard to manage it without a database management system. Advancements in hardware technology enabled personal workstations to run database management systems and create our own knowledge handling systems. Data structure of knowledge is not fixed at the time of the database system development (Kunii, 1987) and this is one of the main reasons why relational models are not suited for representing knowledge. Second reason why relational model fails in representing knowledge is that when number of relationships between relations cross a certain point, it becomes very unintuitive to understand what the relationships are supposed to mean since relationships are implicit in relational models. For a user to query information in a relational model, the user must know the relationships beforehand so that the user can come up with pairs of attributes between relations to connect them. A graph database can represent knowledge more effectively because it can represent complex relationships explicitly.

KNOWLEDGE GRAPHS

Google used the term knowledge graph in 2012 to explain their new way of representing knowledge to their users. Knowledge graphs main goal was adding knowledge depth to information search by representing information through entities, relationships and

attributes (Kejriwal, 2019). This new depth of information representation led to search be about things instead of strings (Singhal, 2012). Google presented three features that they believe made their knowledge graphs valuable:

- Aware of the language's ambiguities and presents what you want.
- Summarizes information effectively.
- Displays relationships and connections deeper than what is searched for.

Knowledge graphs gained popularity fast and quickly started to be adopted by unique domains. One of the domains of knowledge graphs is linked open datasets. DBpedia is one of the most popular linked datasets on the internet. It was first published in 2007 and it is considered a knowledge graph. Most of its data is extracted from Wikipedia and its fundamentals are built on semantic web standards. It uses an RDF graph data model with the SPARQL querying language (Fensel, 2020). DBpedia is an open dataset knowledge graph but there are propriety use cases of knowledge graphs as well. As I've said Google uses knowledge graphs to make their customers life's easier by representing information more effectively. On the other hand, Facebook represents their entity relationship data through knowledge graph which enables them to create social network features that work more efficiently.

Figure 9 Example Knowledge Graph Panel by Google

SOCIAL NETWORKS

Social networks are one of the prime examples of using graph databases because the fundamental information presented by social networks is the relationships among people or groups. These relationships can represent friendship ties, kinship, research networks, etc. Social network analysis is a part of mathematical sociology where analyzing the network can tell us a lot about the social structure of the people and groups.
The main reason relational models are not used for social networks is that they make the network analysis significantly more difficult compared to graph databases. For instance, lets model a friendship tie network between five people using the relational model. First, we will create a people table where we are going to create a single instance for each person such as in

Table 1. Then we can create another table where we define friendships as seen in

Table 2. The first problem with this approach is that it's not natural from a visual perspective. A scientist looking at this data will not gain any information from looking at the friendships table. Second problem is that it makes it difficult to answer the even the simplest questions such as, is there a path of friendships between Bob and Aaron. For a relational model to answer this question, at the worst possible case scenario all of the friendship table needs to be queried to eventually create a graph and run a graph traversal.

Table 1 People Table

Id	Name
1	Bob
2	Alice
3	John
4	Jack
5	Aaron

Table 2 Friendships Table

Person_id_1	Person_id_2
1	2
2	5
3	2
1	4
3	1

For this specific example, the answer to the question of whether there is a path between Bob and Aaron is true. Bob is friends with Alice who is friends with Aaron. Since these tables are small, the computation is not very expensive but if the number of people were on the scale of millions then the number of friendships table would be even larger and the computation would be costly.

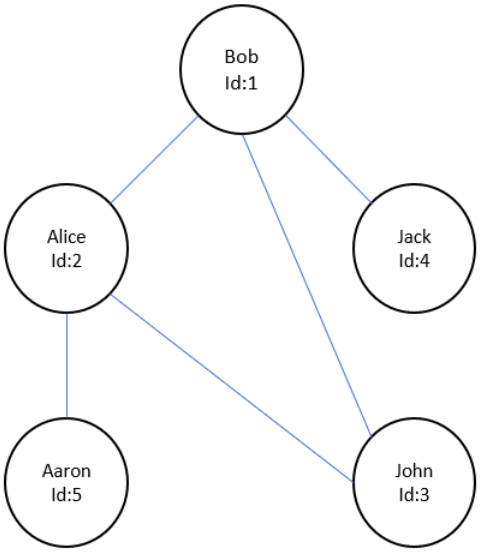

Figure 10 Friendship Graph

```
PREFIX : <http://friendnetwork/>
PREFIX person: <http:// friendnetwork/person/>

ASK { person:1 :isFriend* person:5 }
```

Code Snippet 6 SPARQL Property Path Expression

When we look at this problem from a graph database perspective the computation is rather simple because the graph already exists in the database and only computation that needs to be done is breadth first search on of the person nodes (either Bob or Aaron). For example, SPARQL already has a built-in property path expression functionality. What we mean by property path expression is that, just like string regular expressions, property path expressions can be created to express more generic triple patterns. Code Snippet 6 gives the example SPARQL code that asks whether there is path between person with id 1 and person with id 5. What makes this triple different from the triple patterns described in the previous sections is that the star symbol (*) tries to match the isFriend predicate one or more times. If the pattern is found inside the graph, the ASK

statement is going to return true and else it will return false hence giving us the answer to our question.

Graph Density in Social Networks

Beyond simple path calculations, social networks can give us more insightful sociological information through graph databases. One of the network analyses we can do on a social network is density calculations. The density of a network means the number of connections in a network divided by the number of possible pairs. For instance, in our example friendship network there are five people. If everybody was friends with each other in this network, we would have a total of ten friendships. This is the maximum number of connections we can have in the network and in a fully connected network, the density is 1. In this example there are a total of five friendships out of the possible ten which means the density is 0.5. The sociological meaning of this value can be thought as how closely acquainted a group is in itself. For example, if we are comparing project teams and their performance, we can try to find correlations between the friendship density value and the performance value.

Understanding Power in Social Networks

One of the most important concepts in sociology is power and its significance in social structures. The definition of power in social structures is a debated topic but analyzing social networks from a couple perspectives gives us some ideas on what power is and how to measure it.

Let’s take a look at the star graph in Figure 11 where the edges represent some level of acquaintanceship between the people in the group. If this graph is our only information about the group then it is obvious that Bob is the most powerful person in the

group. Analysis of such networks allowed researchers to come up with definitions on why the central figure is more powerful. There are three concepts that can define power in this structure. First idea is that Bob has higher degree of connection than anybody else in the group meaning he has more opportunities. Bob has four connections in total whereas everybody else has only one. Because of higher chance of opportunities Bob gains power. Second idea is that Bob has higher closeness to everybody in the group. He is only one relationship away from everybody else. On the other hand, everybody else is either two relationships away or a single relationship away. Because he is closer to everybody else it lets Bob's voice to be heard in a larger radius. Last reason is that Bob has higher betweenness than everybody else. For example, if Alice wants to connect with Jack, she has to go through Bob. Bob can use this power to exchange service charges (Hanneman, 2005).

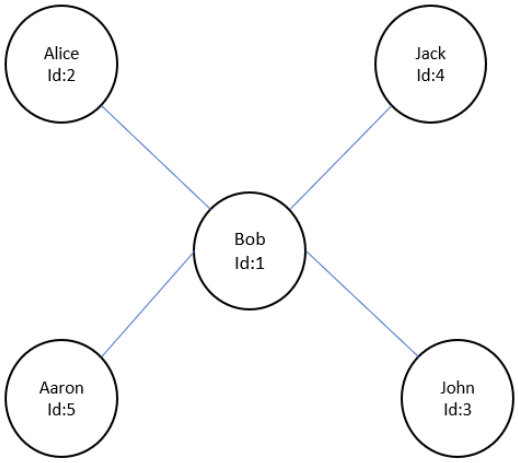

Figure 11 Star Graph

Large Scale Social Network Application Interface: TAO

Facebook is the world's largest social network platform currently consisting of more than 2.6 billion monthly active users ("Facebook 2020 Q1 Results", 2020). Facebook has historically stored its social network data in MySQL databases and queried through PHP

and cached results in memcache (Venkataramani, 2012). Over time, the company built an PHP abstraction that allowed them to query network related data without directly accessing MySQL. "The Associations and Objects" (TAO) is the querying service that was influenced by these PHP abstractions. Its main goal is to create an abstraction layer between the MySQL database and the software engineers that represents the social network data as a graph. TAO was built because the existing PHP abstractions were susceptible to internal failures and they were not usable from non-PHP applications. Facebook's social network focuses on people, actions and relationships (Venkataramani, 2012). TAO graph model represents objects as nodes and associations as edges. Objects are identified by unique id number, object type and data as a number of key value pairs. Associations on the other hand are identified by the two object ids it is connecting and an association type. Figure 12 shows the object and association structure. There could be at most one association of a particular type between two unique objects. The TAO API allows for creation and modification of nodes and associations in addition to querying unique associations and objects.

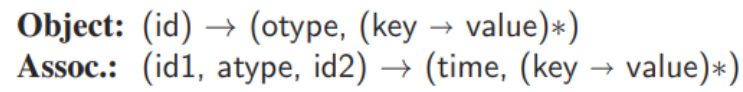

Figure 12 TAO Object and Association Structure (Venkataramani, 2012)

All of the association queries in TAO return association lists. Association lists are identified by having a list of associations which all have the same object id (id1) and association type. Some examples of association queries are getting all associations between id1 and a set of id2's with a particular association type or getting the count of all associations with id1 and with an association type. All of the TAO functions can be mapped to a set of SQL queries that communicate to the underlying MySQL. If in the

future the underlying persistent storage changes from MySQL to another data store like some NoSQL store, TAO can be easily modified to continue having functionality and this would allow all applications that use TAO to not be affected from this storage change. In the previous sections we've talked about one of the shortcomings of relational databases being the inability to horizontally scale. Facebook's TAO infrastructure solves this problem through database sharding. Database sharding is when each database server is divided into one or more *shards*. Shards are abstracted virtual pieces of a database and they allow for easier data search when the shard id is already known. In TAO design each object id in itself contains the shard id, which shows the database server the object belongs to. Associations are stored in the same shard as their object1. By internally including the shard id in the object data, the object search becomes significantly faster because the querying applications does not need to look at every single server in the case of a distributed database server.

Overall TAO has been a very successful service in serving Facebook's many social network applications because not only does it create an expandable and efficient querying interface it also creates the abstraction of a graph that is accessible through various software applications.

GRAPH MINING

As explained the previous chapters, networks and graphs can be used to model various data structures and real-world scenarios. Graph mining is a set of techniques to examine and generate new information from these graph structures. Some of this new information can be centered around prediction tasks like can there be a link between a pair of nodes (link prediction) or finding communities within nodes (Arsov, 2019). One way to answer these predication questions is through applying machine learning algorithms on these

graphs. To apply machine learning algorithms on graphs, a set of new techniques called network embeddings and knowledge graph embeddings have been created. These techniques allow a set of nodes or edges to be represented as low dimensional feature vectors which then can be fed into existing machine learning models to be used for classification or regression tasks. Overall graph mining algorithms can be utilized in almost all of the graph database applications where we want to uncover embedded meanings in a database and they are crucial for the advancement of artificial intelligence applications.

Conclusion

Relational database models have long succeeded in being the primary choice of storage model in the last fifty years. They are very good at representing relatively linear data where the connections are not too complex. With the rise of web and semi structured data, highly connected and complex data ended up being more and more prevalent in the technology world. Relational models have failed at representing highly connected data and this resulted in a need for new database models. Various database models like object-oriented models, key value stores and graph-based models emerged from this new need. Graph based models especially have shown their strengths in certain domains like web, social networks and transport infrastructures. Various technology companies and independent software developers have developed their own implementations of graph stores such as Neo4j, ArangoDB and OrientDB. Graph database technologies have contributed greatly on the growth of science and technology and they will continue to contribute for the foreseeable future.

References

Abiteboul, S. (1970). Querying Semi-Structured DataProc. ICDT 97.

Angles, R. (2018). The Property Graph Database Model. AMW.

Angles, R., & Gutierrez, C. (2008). Survey of Graph Database Models. ACM Comput. Surv., 40(1).

Arsov, N., & Mirceva, G. (2019). Network Embedding: An Overview. *ArXiv, abs/1911.11726*.

Codd, E. (1970). A Relational Model of Data for Large Shared Data Banks. Commun. ACM, 13(6), 377–387.

Curé, Olivier, and Guillaume Blin. (2015). RDF Database Systems Triples Storage and SPARQL Query Processing. Morgan Kaufmann Publishers.

Czerepicki, A. (2016). Application of graph databases for transport purposes. Bulletin of the Polish Academy of Sciences Technical Sciences, 64.

Davoudian, A., Chen, L., & Liu, M. (2018). A Survey on NoSQL Stores. ACM Comput. Surv., 51(2).

Facebook Reports First Quarter 2020 Results. (2020). Retrieved August 14, 2020, from https://investor.fb.com/investor-news/press-release-details/2020/Facebook-Reports-First-Quarter-2020-Results/default.aspx

Fensel, D. (2020). *Knowledge graphs: Methodology, tools and selected use cases*. Cham, Switzerland: Springer.

Hanneman, R., & Riddle, M. (2005). Introduction to Social Network Methods. Department of Sociology, University of California, Riverside.

Harary, F. (2001). Graph theory (pp. 1-2) (F. Harary, Author). Cambridge, MA: Perseus Books.

Hurlburt, G., Thiruvathukal, G., & Lee, M. (2017). The Graph Database: Jack of All Trades or Just Not SQL? IT Professional, 19, 21-25.

Jatana, N., Puri, S., Ahuja, M., Kathuria, I., & Gosain, D. (2012). A Survey and Comparison of Relational and Non-Relational Database. *International journal of engineering research and technology, 1*.

Kejriwal, M. (2019). *Domain-specific knowledge graph construction*. Cham, Switzerland: Springer.

Kim, Won. (1990). Object-oriented databases: definition and research directions. IEEE Transactions on Knowledge and Data Engineering, 2(3), 327-341.

Kunii, H. (1987). DBMS with Graph Data Model for Knowledge Handling. In Proceedings of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow (pp. 138–142). IEEE Computer Society Press.

OrientDB Manual. (2020). Retrieved August 14, 2020, from https://orientdb.com/docs/3.0.x/

Papakonstantinou, Y., Garcia-Molina, H., & Widom, J. (1995). Object Exchange Across Heterogeneous Information Sources. In Proceedings of the Eleventh International Conference on Data Engineering (pp. 251–260). IEEE Computer Society.

Pérez, C. (2006). Semantics and Complexity of SPARQL. In The Semantic Web - ISWC 2006 (pp. 30–43). Springer Berlin Heidelberg.

Pokorný, J. (2013). NoSQL databases: a step to database scalability in web environment. *Int. J. Web Inf. Syst., 9*, 69-82.

Sasaki, B. M. (2018, July 17). Graph Databases for Beginners: Why Graph Technology Is the Future. Retrieved August 14, 2020, from https://neo4j.com/blog/why-graph-databases-are-the-future/

Schaefer, L. (2020). What is NoSQL? NoSQL Databases Explained. Retrieved August 14, 2020, from https://www.mongodb.com/nosql-explained

Silberschatz, A., Korth, H., & Sudarshan, S. (1996). Data Models. ACM Comput. Surv., 28(1), 105–108.

Singhal, A. (2012, May 16). Introducing the Knowledge Graph: Things, not strings. Retrieved August 14, 2020, from https://www.blog.google/products/search/introducing-knowledge-graph-things-not/

Venkataramani, V., Amsden, Z., Bronson, N., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Hoon, J., Kulkarni, S., Lawrence, N., Marchukov, M., Petrov, D., & Puzar, L. (2012). TAO: how facebook serves the social graph. SIGMOD Conference.

What is a Graph Database? (2020). Retrieved August 14, 2020, from https://neo4j.com/developer/graph-database/

What is a Multi-model Database and Why Use It - ArangoDB White Paper. (2020, April). Retrieved August 14, 2020, from https://www.arangodb.com/arangodb-white-papers/white-paper-multi-model-database/

World Wide Web Consortium. 2014b. RDF 1.1 Primer.

Wylot, M., Hauswirth, M., Cudré-Mauroux, P., & Sakr, S. (2018). RDF Data Storage and Query Processing Schemes: A Survey. ACM Comput. Surv., 51(4).